US 20120137178A1

(19) United States

(12) Patent Application Publication

BARSNESS et al.

(10) Pub. No.: US 2012/0137178 A1

(43) Pub. Date: May 31, 2012

(54) STREAM BASED DEBUGGING TECHNIQUES

(75) Inventors: **ERIC L. BARSNESS**, PINE ISLAND, MN (US); **RYAN K. CRADICK**, ORONOCO, MN (US); **MICHAEL D. PFEIFER**, ROCHESTER, MN (US); **JOHN M. SANTOSUOSSO**, ROCHESTER, MN (US)

(73) Assignee: **INTERNATIONAL BUSINESS MACHINES CORPORATION**, ARMONK, NY (US)

(21) Appl. No.: **12/956,424**

(22) Filed: **Nov. 30, 2010**

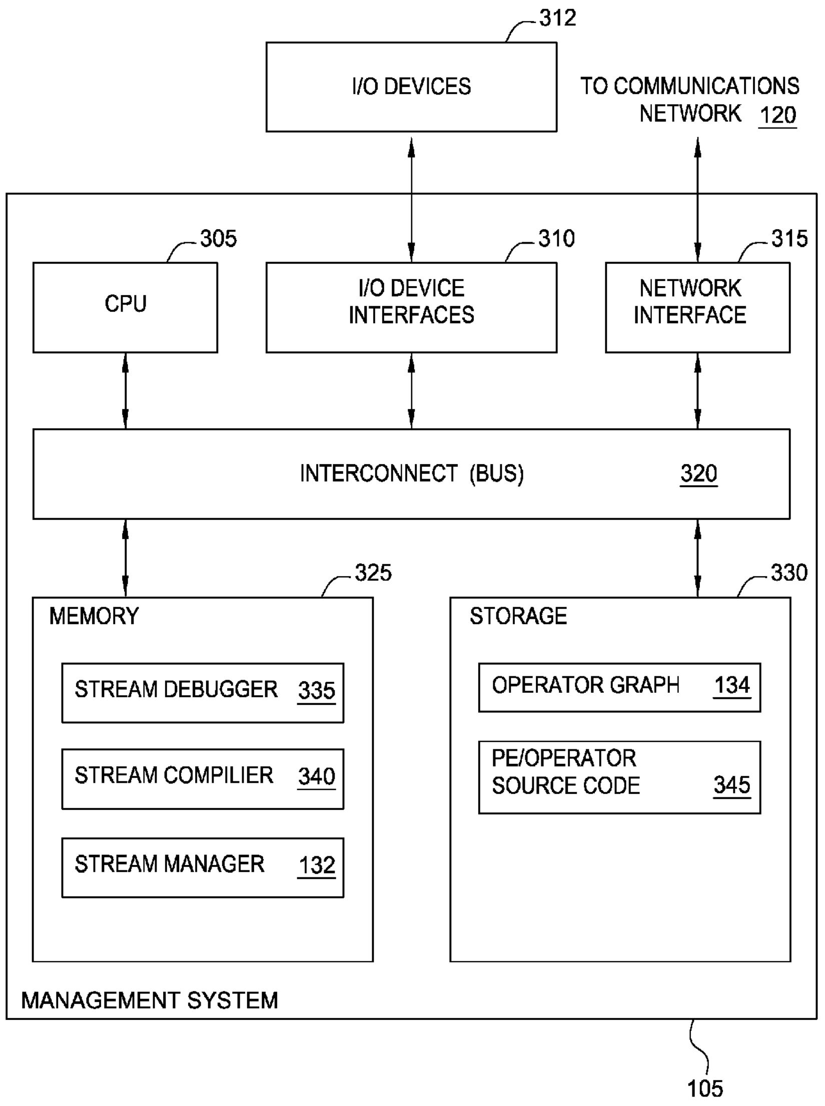
Publication Classification

(51) **Int. Cl.**
G06F 11/36 (2006.01)

(52) **U.S. Cl.** **714/35**; 714/E11.21

(57) **ABSTRACT**

Techniques are described for debugging a processing element (or elements) in a stream based database application in a manner that reduces the impact of debugging the processing element (or elements) on the overall running environment by selectively fusing (or un-fusing) processing elements running on a group of compute nodes. In addition to fusing and un-fusing processing elements or otherwise modifying a state of the stream application, a debugging application and stream manager may modify data flows within the application stream in a variety of ways to minimize any disruption resulting from a debugging session.



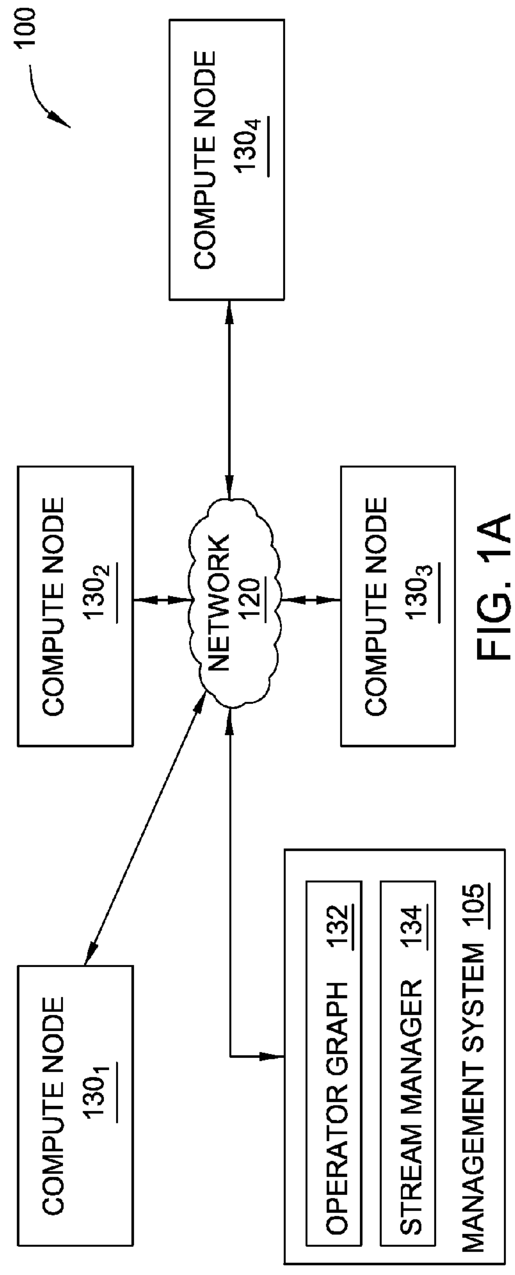

100
COMPUTE NODE 130_4
COMPUTE NODE 130_2
NETWORK 120
COMPUTE NODE 130_3
COMPUTE NODE 130_1
OPERATOR GRAPH 132
STREAM MANAGER 134
MANAGEMENT SYSTEM 105

FIG. 1A

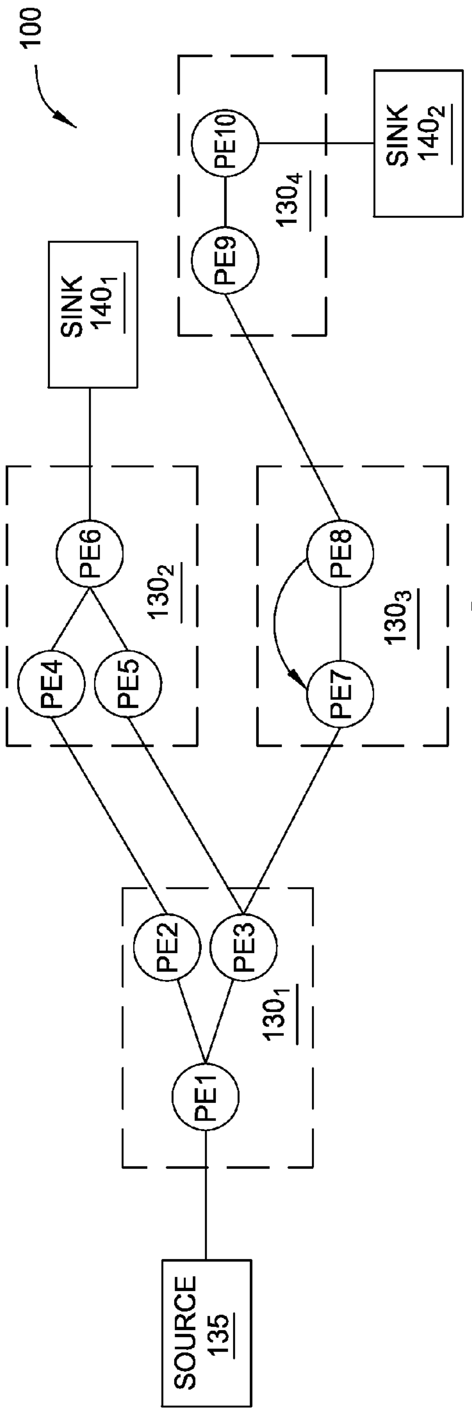

100
SINK 140_2
PE10
PE9
130_4
SINK 140_1
PE6
PE4
PE5
130_2
PE8
PE7
130_3
PE2
PE3
PE1
130_1
SOURCE 135

FIG. 1B

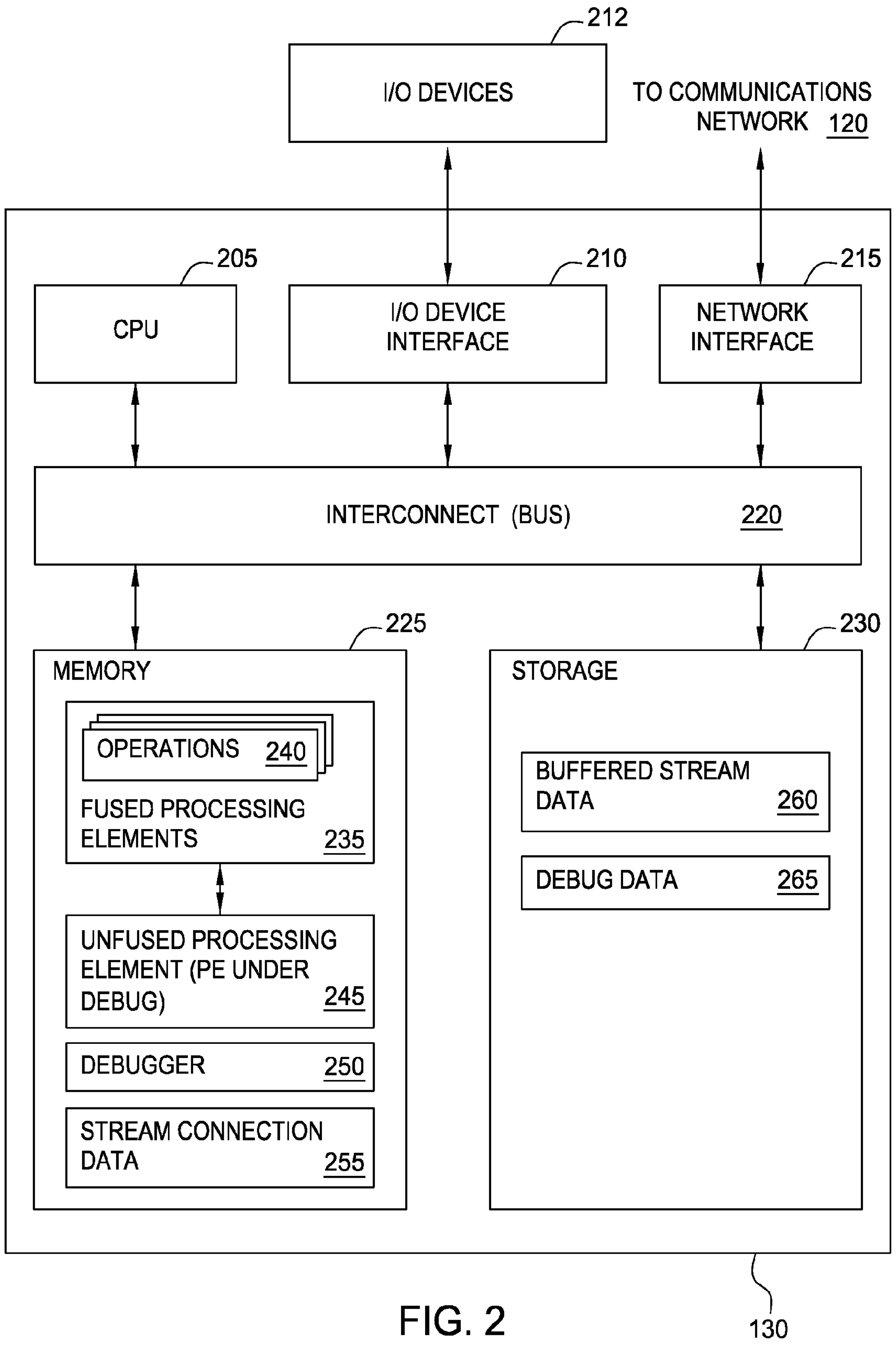

212
I/O DEVICES
TO COMMUNICATIONS NETWORK 120
205
CPU
210
I/O DEVICE INTERFACE
215
NETWORK INTERFACE
INTERCONNECT (BUS) 220
225
MEMORY
OPERATIONS 240
FUSED PROCESSING ELEMENTS 235
UNFUSED PROCESSING ELEMENT (PE UNDER DEBUG) 245
DEBUGGER 250
STREAM CONNECTION DATA 255
230
STORAGE
BUFFERED STREAM DATA 260
DEBUG DATA 265
130

FIG. 2

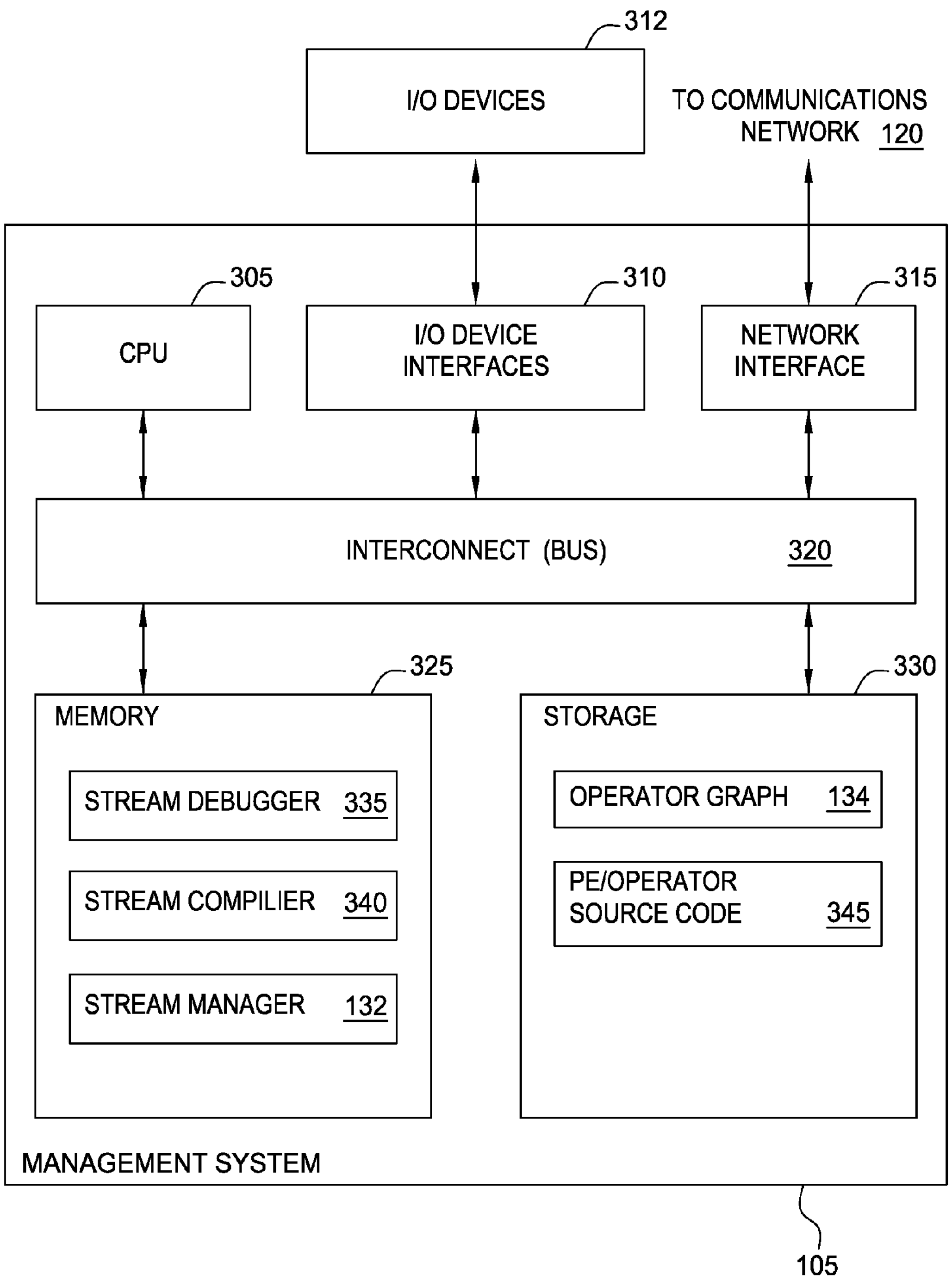

312
I/O DEVICES
TO COMMUNICATIONS NETWORK 120
305
CPU
310
I/O DEVICE INTERFACES
315
NETWORK INTERFACE
INTERCONNECT (BUS) 320
325
MEMORY
STREAM DEBUGGER 335
STREAM COMPILIER 340
STREAM MANAGER 132
330
STORAGE
OPERATOR GRAPH 134
PE/OPERATOR SOURCE CODE 345
MANAGEMENT SYSTEM
105

FIG. 3

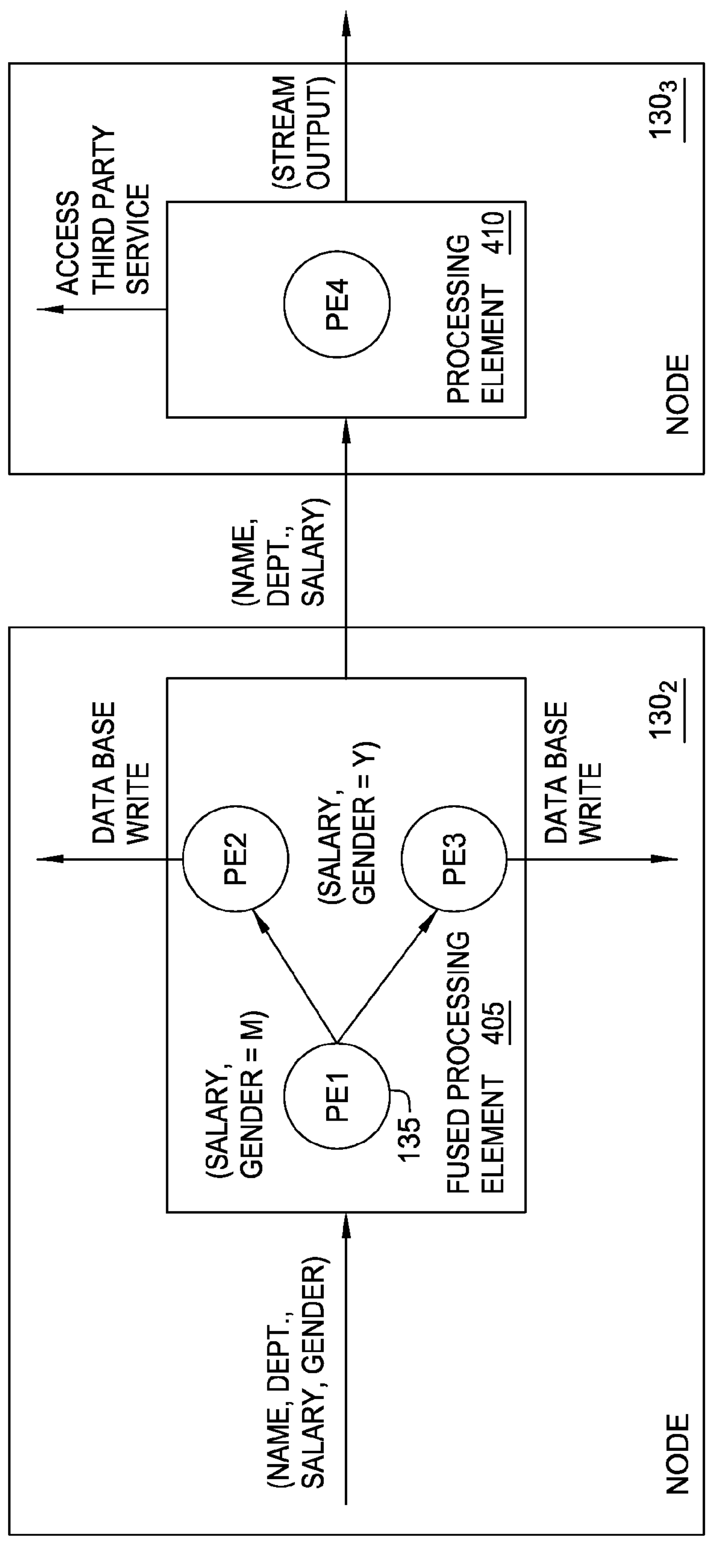

(NAME, DEPT., SALARY, GENDER)
(SALARY, GENDER = M)
PE1
135
FUSED PROCESSING ELEMENT 405
PE2
PE3
(SALARY, GENDER = Y)
DATA BASE WRITE
DATA BASE WRITE
NODE
130_2
(NAME, DEPT., SALARY)
ACCESS THIRD PARTY SERVICE
PE4
PROCESSING ELEMENT 410
(STREAM OUTPUT)
NODE
130_3

FIG. 4

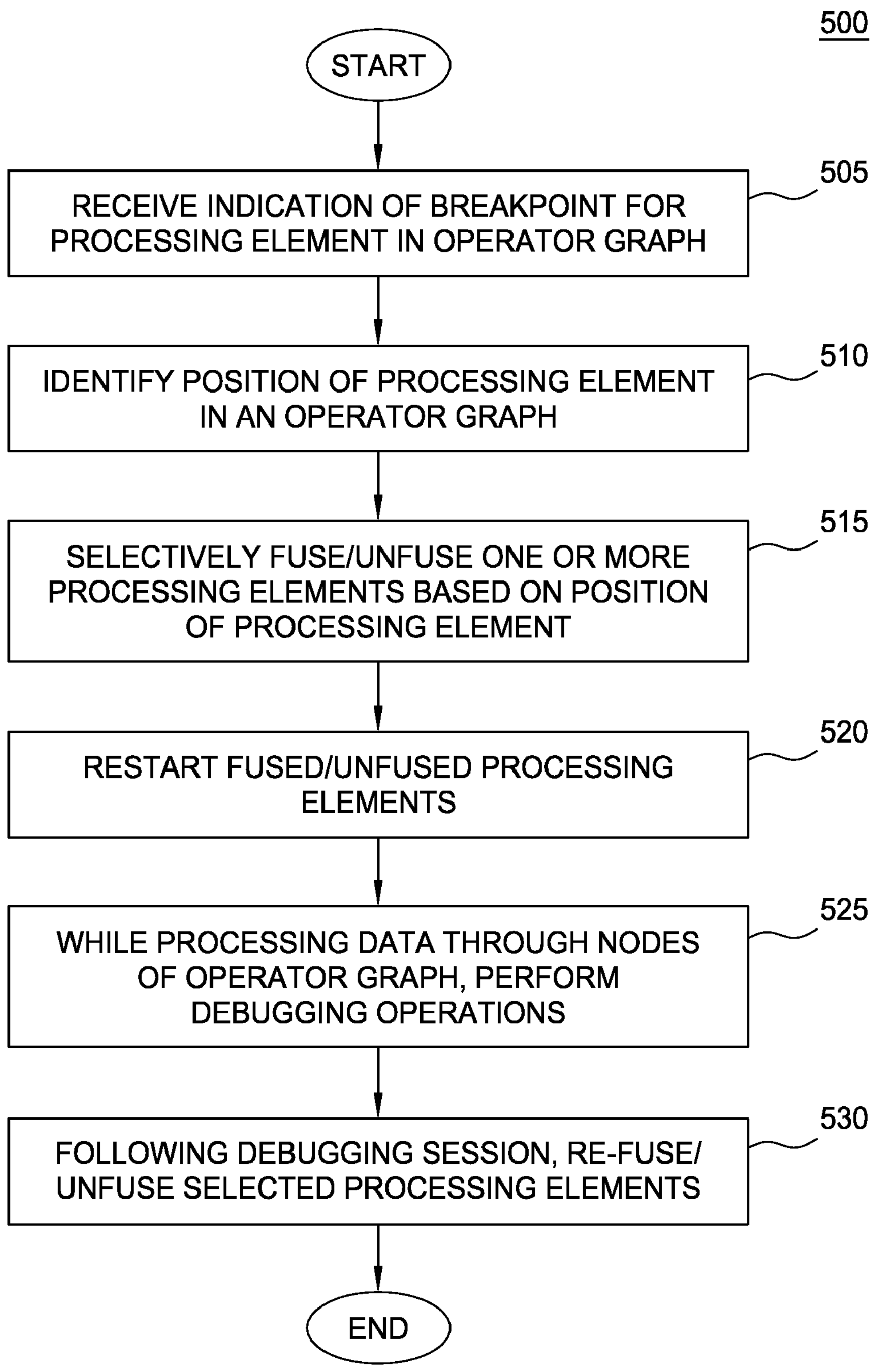

500
START
RECEIVE INDICATION OF BREAKPOINT FOR PROCESSING ELEMENT IN OPERATOR GRAPH
505
IDENTIFY POSITION OF PROCESSING ELEMENT IN AN OPERATOR GRAPH
510
SELECTIVELY FUSE/UNFUSE ONE OR MORE PROCESSING ELEMENTS BASED ON POSITION OF PROCESSING ELEMENT
515
RESTART FUSED/UNFUSED PROCESSING ELEMENTS
520
WHILE PROCESSING DATA THROUGH NODES OF OPERATOR GRAPH, PERFORM DEBUGGING OPERATIONS
525
FOLLOWING DEBUGGING SESSION, RE-FUSE/ UNFUSE SELECTED PROCESSING ELEMENTS
530
END

FIG. 5

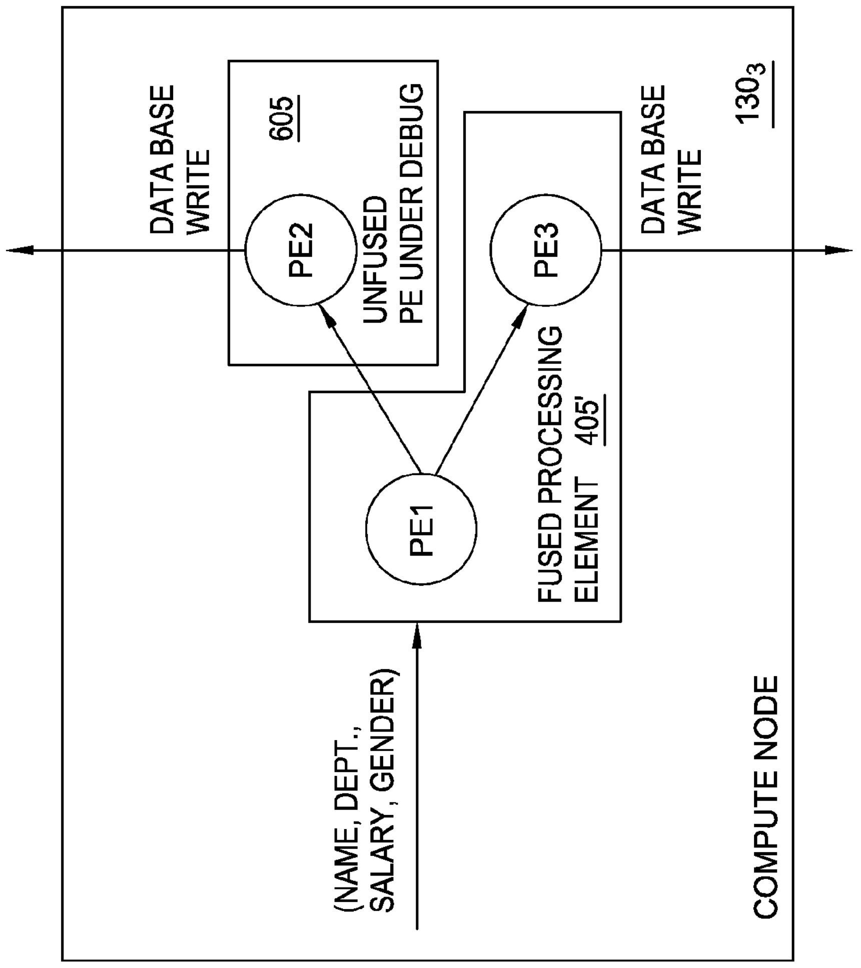

DATA BASE WRITE
605
PE2
UNFUSED PE UNDER DEBUG
PE3
DATA BASE WRITE
130_3
PE1
FUSED PROCESSING ELEMENT 405'
(NAME, DEPT., SALARY, GENDER)
COMPUTE NODE

FIG. 6

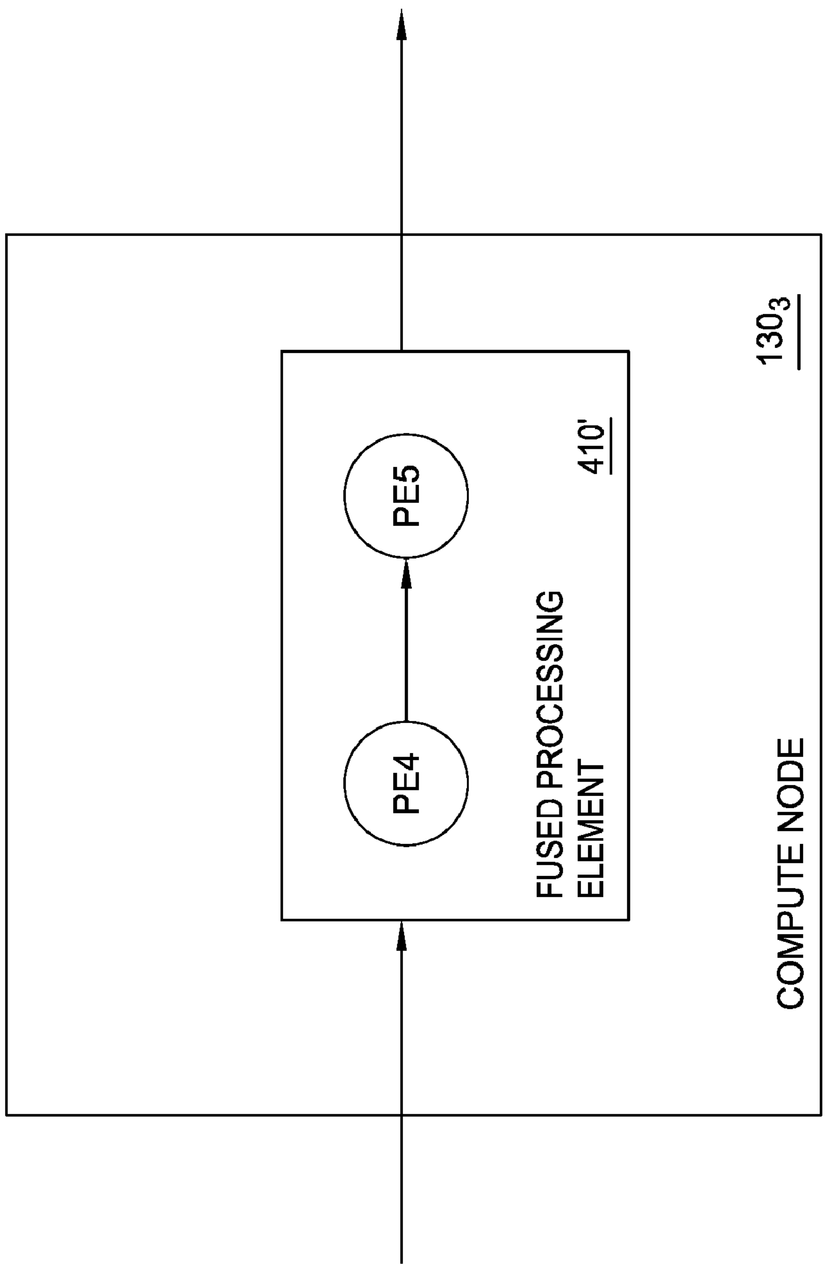


FIG. 7

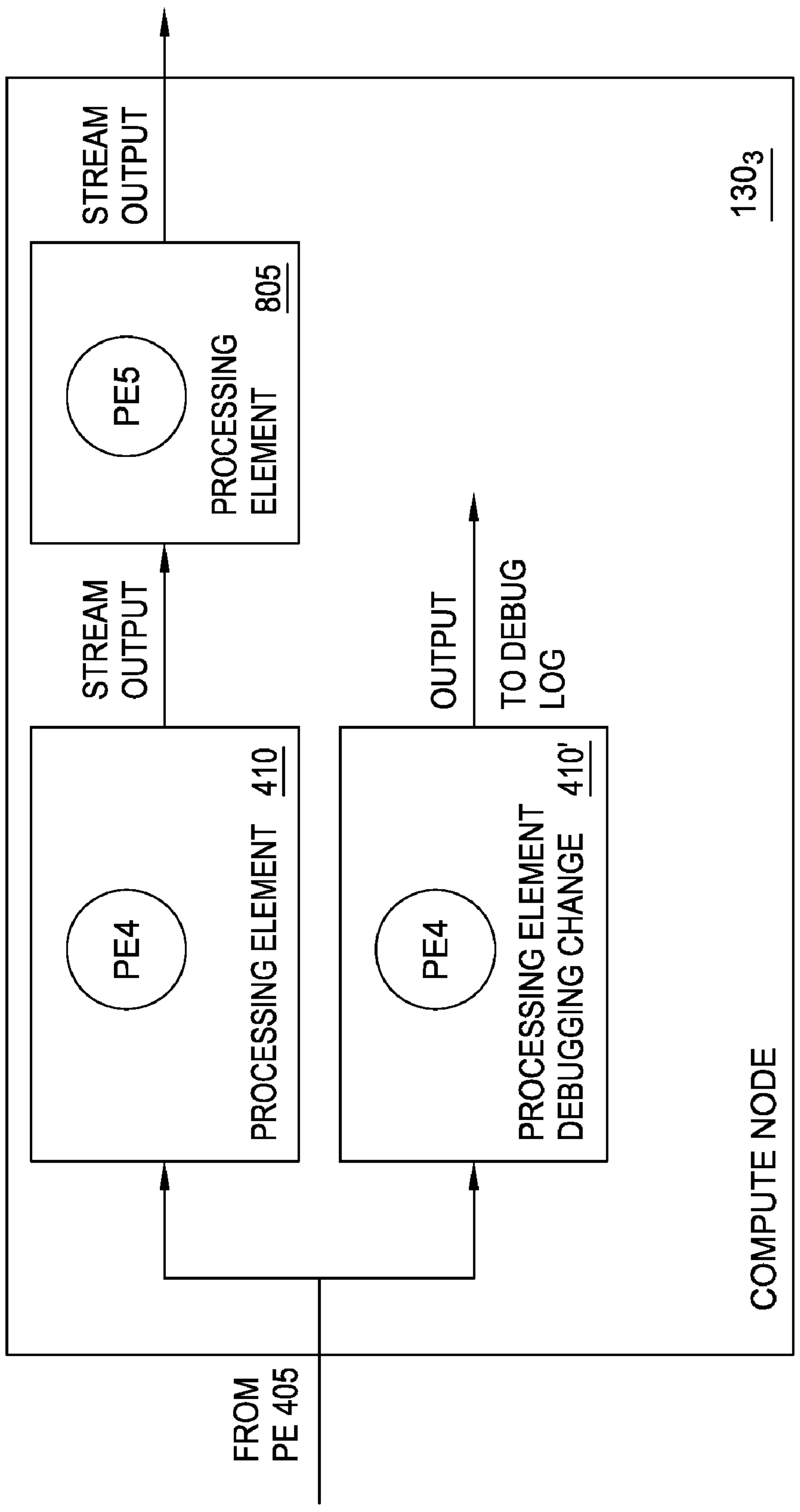

FROM PE 405
COMPUTE NODE
PE4
PROCESSING ELEMENT 410
PE4
PROCESSING ELEMENT DEBUGGING CHANGE 410'
STREAM OUTPUT
OUTPUT TO DEBUG LOG
PE5
PROCESSING ELEMENT 805
STREAM OUTPUT
130_3

FIG. 8

STREAM BASED DEBUGGING TECHNIQUES

BACKGROUND

[0001] While computer databases have become extremely sophisticated, the computing demands placed on database systems have increased at a rapid pace. Database systems are typically configured to separate the process of storing data from accessing, manipulating or using data stored in the database. More specifically, databases use a model where data is first stored, then indexed, and then queried. However, this model cannot meet the performance requirements of some real-time applications. For example, the rate at which a database system can receive and store incoming data can limit how much data can be processed or otherwise evaluated, which, in turn, limits the utility of database applications configured to process large amounts of data in real-time.

[0002] To address this issue, stream based computing and stream based database computing is emerging as a developing technology for database systems. And products are available which allow users to create applications that process and query streaming data before it reaches a database file. With this emerging technology, users can specify processing logic to apply to inbound data records while they are "in flight," with the results available in milliseconds. Constructing an application using this type of processing has opened up a new programming paradigm that will allow for a broad variety of innovative applications, systems and processes to be developed as well as present new challenges for application programmers and database developers.

SUMMARY

[0003] Embodiments of the invention provide techniques for debugging a processing element (or elements) in a stream based database application. For example, one embodiment of the invention includes a method of debugging a stream application composed from a plurality of processing elements executing on one or more compute nodes. The method may generally include receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements and evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality. This method may also include modifying a state of the stream application to allow the first processing element to be debugged and initiating a debug session for the first processing element.

[0004] Another embodiment of the invention includes a computer-readable storage medium containing a program which, when executed, performs an operation for debugging a stream application composed from a plurality of processing elements executing on one or more compute nodes. The operation itself may generally include receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements and evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality. The operation may further include modifying a state of the stream application to allow the first processing element to be debugged and initiating a debug session for the first processing element.

[0005] Still another embodiment of the invention includes a system having a plurality of compute nodes, each including a processor and a memory. The compute nodes are configured to execute processing elements of a stream application. The system may also include a management system also having a processor and a memory. The memory stores a stream debugging application, which, when executed on the management system, is configured to perform an operation for debugging the stream application executing on the plurality of compute nodes.

[0006] The operation itself may generally include receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements and evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality. The operation may further include modifying a state of the stream application to allow the first processing element to be debugged and initiating a debug session for the first processing element.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] So that the manner in which the above recited aspects are attained and can be understood in detail, a more particular description of embodiments of the invention, briefly summarized above, may be had by reference to the appended drawings. Note, however, that the appended drawings illustrate only typical embodiments of this invention and are therefore not to be considered limiting of its scope, for the invention may admit to other equally effective embodiments.

[0008] FIGS. **1**A-**1**B illustrate a computing infrastructure configured to execute a stream database application, according to one embodiment of the invention.

[0009] FIG. **2** is a more detailed view of a distributed computing node of FIG. **1**, according to one embodiment of the invention.

[0010] FIG. **3** is a more detailed view of a management computing system in a stream database application, according to one embodiment of the invention.

[0011] FIG. **4** illustrates an example of compute nodes in a stream database application, according to one embodiment of the invention.

[0012] FIG. **5** illustrates a method for debugging a processing element in a stream database application, according to one embodiment of the invention.

[0013] FIG. **6**-**8** provide examples of the compute nodes and stream application first shown in FIG. **4** modified to illustrate scenarios for debugging the stream application, according to embodiments of the invention.

DETAILED DESCRIPTION

[0014] In a stream application, operators are connected to one another such that data flows from one processing element to the next (e.g., over a TCP/IP socket). Scalability is reached by distributing an application across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them. Processing elements (and operators) in a stream application can be fused together to form a larger processing element. Doing so allows processing elements to share a common process space, resulting in much faster communication between operators than is available using interprocess communication techniques (e.g., using a TCP/IP socket). Further, processing elements can be inserted or removed dynamically from an operator graph representing the flow of data through the stream application, as well as fused or un-fused from a stream application during runtime.

[0015] While processing elements in a distributed stream environment may be debugged by reviewing log files generated by running code, frequently a complete debug session is needed. However, running a debugger slows down a process being debugged, regardless of whether a breakpoint is hit, and triggering a breakpoint can result in a substantial slowdown (if not complete halt) to data flowing through an operator under debug. Thus, running a processing element under debug can cause "stream backup," as data streams flowing to the operators in that processing element do not stop flowing. More generally, stream computing is based on the premise that streams continuously flow through the operators.

[0016] When multiple processing elements are fused together to form a single running process, triggering a break point in one causes all of the fused processing elements to stop processing, potentially turning off a large portion of an operator graph. On the other hand, a set of un-fused processing elements may include a first processing element configured to join data streamed from a second and third processing element. If a breakpoint is triggered in either the second or third processing element, the stream processing results can be disrupted.

[0017] Embodiments of the invention provide techniques for debugging a processing element (or elements) in a stream based application. In particular, embodiments of the invention provide techniques for modifying a state of the stream application during debugging a processing element in a manner that reduces the impact of debugging the processing element (or elements) on the overall running environment. For example, assume that one operator within a fused processing element of twenty operators needs to be debugged. In such a case, a stream debugger may be configured to remove the fused processing element from the running stream, un-fuse the operators, and re-insert them into the running stream. The processing element with the single operator being debugged may then execute as an independently running process (with its own process ID (PID) and memory space), which allows that operator to be debugged independently from the other nineteen. Once the debugging session is complete, the debugger may re-fuse the un-fused operator, restoring the processing element of twenty fused operators to the same running state that existed prior to debug session.

[0018] In another embodiment, the debugger may fuse multiple independent processing elements together as part of a debug session. This may be useful to halt a portion of an operator graph from running when one operator is being debugged. For example, doing so may allow any dependencies between processing elements to be satisfied. Similarly, the debugger may in appropriate cases prevent data from flowing into the stream while a given processing element is being debugged. This can occur at any point upstream (or downstream) from a processing element in the operator graph being debugged. More specifically, the debugger may prevent data from flowing to a source operator (i.e., a processing element that is the source point for data flowing through an operator graph) or from flowing to a specific part (or specific processing element) in the operator graph. Conversely, the debugger may be configured to recognize some tuples of incoming data as being exempt from a breakpoint.

[0019] As an alternative to preventing data from flowing, data may be "load shed" at a processing element under debug (or selectively allowed to flow under some conditions). Doing so may help avoid overloading the system with data when the debugging session is complete; particularly in cases where data may not have value if not processed promptly. In still another embodiment, the debugger may be configured to duplicate a processing element such that debugging occurs in the duplicated one. Data flowing out of the debugged processing element may be shed instead of being sent to the linked, downstream operators. Further, the results output by the operator under debug may be compared to the real operator as a check on the accuracy of the debugging process.

[0020] In the following, reference is made to embodiments of the invention. However, it should be understood that the invention is not limited to specific described embodiments. Instead, any combination of the following features and elements, whether related to different embodiments or not, is contemplated to implement and practice the invention. Furthermore, although embodiments of the invention may achieve advantages over other possible solutions and/or over the prior art, whether or not a particular advantage is achieved by a given embodiment is not limiting of the invention. Thus, the following aspects, features, embodiments and advantages are merely illustrative and are not considered elements or limitations of the appended claims except where explicitly recited in a claim(s). Likewise, reference to "the invention" shall not be construed as a generalization of any inventive subject matter disclosed herein and shall not be considered to be an element or limitation of the appended claims except where explicitly recited in a claim(s).

[0021] As will be appreciated by one skilled in the art, aspects of the present invention may be embodied as a system, method or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system." Furthermore, aspects of the present invention may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

[0022] Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable signal medium or a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium would include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain, or store a program for use by or in connection with an instruction execution system, apparatus or device.

[0023] A computer readable signal medium may include a propagated data signal with computer readable program code embodied therein, for example, in baseband or as part of a carrier wave. Such a propagated signal may take any of a variety of forms, including, but not limited to, electro-magnetic, optical, or any suitable combination thereof. A computer readable signal medium may be any computer readable medium that is not a computer readable storage medium and that can communicate, propagate, or transport a program for use by or in connection with an instruction execution system, apparatus, or device.

[0024] Program code embodied on a computer readable medium may be transmitted using any appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

[0025] Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language such as Java, Smalltalk, C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

[0026] Aspects of the present invention are described below with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0027] These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

[0028] The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0029] Embodiments of the invention may be provided to end users through a cloud computing infrastructure. Cloud computing generally refers to the provision of scalable computing resources as a service over a network. More formally, cloud computing may be defined as a computing capability that provides an abstraction between the computing resource and its underlying technical architecture (e.g., servers, storage, networks), enabling convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Thus, cloud computing allows a user to access virtual computing resources (e.g., storage, data, applications, and even complete virtualized computing systems) in "the cloud," without regard for the underlying physical systems (or locations of those systems) used to provide the computing resources.

[0030] Typically, cloud computing resources are provided to a user on a pay-per-use basis, where users are charged only for the computing resources actually used (e.g., an amount of storage space consumed by a user or a number of virtualized systems instantiated by the user). A user can access any of the resources that reside in the cloud at any time, and from anywhere across the Internet. In context of the present invention, a user may access applications or related data available in the cloud. For example, the nodes used to create a stream database application may be virtual machines hosted by a cloud service provider.

[0031] FIGS. **1**A-**1**B illustrate a computing infrastructure **100** configured to execute a stream application, according to one embodiment of the invention. As shown, the computing infrastructure **100** includes a management system **105** and a plurality of compute nodes **130**$_{1-4}$, each connected to a communications network **120**. Also, the management system **105** includes an operator graph **132** and a stream manager **134**. As described in greater detail below, the operator graph **132** represents a stream application beginning from of one or more source processing elements (PEs) through to one or more sink PEs. Data elements flow into a source PE of a stream application and are processed by that PE. Typically, processing elements receive an N-tuple of data elements from the stream as well as emit an N-tuple of data elements into the stream (except for a sink PE where the stream terminates). Of course, the N-tuple received by a processing element need not be the same N-tuple sent downstream. And the processing elements could be configured to receive or emit data in formats other than an N-tuple (e.g., the processing elements could exchange data marked up as XML documents). Additionally, each processing element may be configured to carry out any form of data processing functions on the received tuple, including, e.g., writing to database tables or performing other database operations such as data joins, splits, reads, etc., as well as performing other data analytic functions or operations.

[0032] The stream manager **134** may be configured to monitor a stream application running on the compute nodes **130**$_{1-4}$ as well as change the structure of the operator graph **134**. For example, the stream manager **134** may move processing elements (PEs) from one compute node **130** to another, e.g., to manage the processing loads of the compute nodes **130** in the computing infrastructure **100**. Further, stream manager **134** may control the stream application by inserting, removing, fusing, un-fusing, or otherwise modifying the processing elements (or what data-tuples flow to the processing elements) running on the compute nodes **130**$_{1-4}$.

[0033] FIG. **1**B illustrates an example operator graph that includes ten processing elements (labeled as PE**1**-PE**10**) running on the compute nodes **130**$_{1-4}$. While a processing element may be executed as an independently running process (with its own process ID (PID) and memory space), multiple processing elements may be fused to run as single process (with a PID and memory space). In cases where two (or more) processing elements are running independently, inter-process communication may occur using a network socket (e.g., a TCP/IP socket). However, when processes are fused together, the fused processing elements can use more rapid communication techniques for passing N-tuples (or other data) among processing elements (and operators in each processing element).

[0034] As shown, the operator graph begins at a source PE **135** (labeled as PE**1**) and ends at sink PEs $\mathbf{140}_{1-2}$ (labeled as PE**6** and PE**10**). Compute node $\mathbf{130}_1$ includes source PE**1** along with PE**2** and PE**3**. Source PE**1** emits tuples received by PE**2** and PE**3**. For example, PE**1** may split data elements received in a tuple and pass some data elements to PE**2** others to PE**3**. Data that flows to PE**2** results in tuples emitted to PE**4** on compute node $\mathbf{130}_2$. And data tuples emitted by PE**4** flow to sink PE**6** $\mathbf{140}_1$. Similarly, data tuples flowing from PE**3** to PE**5** also reach sink PE**6** $\mathbf{140}_1$. Thus, in addition to being a sink for this example operator graph, PE**6** could be configured to perform a join operation, combining tuples received from PE**4** and PE**5**. This example operator graph also shows data tuples flowing from PE**3** to PE**7** on compute node $\mathbf{130}_3$, which itself shows data tuples flowing to PE**8** and looping back to PE**7**. Data tuples emitted from PE**8** flow to PE**9** on compute node $\mathbf{130}_4$, which emits tuples processed by sink PE**10** $\mathbf{140}_2$.

[0035] FIG. **2** is a more detailed view of the compute node **130** of FIGS. **1**A-**1**B, according to one embodiment of the invention. As shown, the compute node **130** includes, without limitation, a central processing unit (CPU) **205**, a network interface **215**, an interconnect **220**, a memory **225**, and storage **230**. The compute node **130** may also include an I/O devices interface **210** used to connect I/O devices **212** (e.g., keyboard, display and mouse devices) to the compute node **130**.

[0036] The CPU **205** retrieves and executes programming instructions stored in the memory **225**. Similarly, the CPU **205** stores and retrieves application data residing in the memory **225**. The interconnect **220** is used to transmit programming instructions and application data between the CPU **205**, I/O devices interface **210**, storage **230**, network interface **215**, and memory **225**. CPU **205** is included to be representative of a single CPU, multiple CPUs, a single CPU having multiple processing cores, and the like. And the memory **225** is generally included to be representative of a random access memory. Storage **230**, such as a hard disk drive, solid state device (SSD), or flash memory storage drive, may store non-volatile data.

[0037] In this example, the memory **225** includes a fused processing element (PE) **235**, an un-fused PE **245**, a debugger application **250**, and stream connection data **255**. The fused PE **235** includes a collection of operators **240**. As noted above, each operator **240** may provide a small chunk of executable code configured to process data flowing into a processing element (e.g., PE **235**) and to emit data to other operators **240** in that PE and to other PEs in the stream application. Such PEs may be on the same compute node **130** (e.g., un-fused PE **245**) or on other compute nodes accessed over the data communications network **120**. The stream connection data **255** represents the connections between PEs on compute node **130** (e.g., a TCP/IP socket connection between the fused PE **240** and un-fused PE **245**), as well as connections to other compute nodes **130** with upstream and or downstream PEs in the stream application, also via TCP/IP sockets (or other inter-process data communication mechanisms).

[0038] Buffered stream data **260** represents a storage space for data flowing into the compute node **105** from upstream processing elements (or from a data source for the stream application). For example, buffered stream data may include data tuples waiting to be processed by one of the PEs **240** or **245**. Buffered stream data **260** may also store the results of data processing performed by PEs **240** or **245** that will be sent to downstream processing elements (or load shed).

[0039] The debugger **250** provides a software application configured to allow a developer to debug processing element **245** running on the compute node **130**. For example, the debugger **250** may be used to set breakpoints, to execute instructions step into (or over) function calls in the instructions, to inspect variables, etc., as well as provide a variety of other functions and/or features used for debugging the processing element **245**. In context of the present invention, the debugger **250** may be configured to selectively fuse and un-fuse PEs (or otherwise modify a state of the stream application) to facilitate the debugging process. In one embodiment, e.g., the debugger may fuse and un-fuse PEs as specified by the developer. Alternatively, the debugger **250** may be configured to propose a stream state for debugging a given processing element (or elements). To do so, the debugger **250** may interact with a stream debugger on the management system **130** to analyze an operator graph as well as a distribution of processing elements across a collection of compute nodes **105** in order to determine how to modify the operator graph associated with a stream application. In addition to fusing and un-fusing processing elements, the debugger **250** may modify the running state of the stream application by, e.g., halting (or limiting) data flows across a PE being debugged, specifying data exempt from the debugging process, duplicating PEs to create a debug copy of a PE, or performing other actions to allow a processing element (or elements) to be debugged.

[0040] FIG. **3** is a more detailed view of the server computing system **105** of FIG. **1**, according to one embodiment of the invention. As shown, server computing system **105** includes, without limitation, a central processing unit (CPU) **305**, a network interface **315**, an interconnect **320**, a memory **325**, and storage **330**. The client system **130** may also include an I/O device interface **310** connecting I/O devices **312** (e.g., keyboard, display and mouse devices) to the server computing system **105**.

[0041] Like CPU **205** of FIG. **2**, CPU **305** is configured to retrieve and execute programming instructions stored in the memory **325** and storage **330**. Similarly, the CPU **305** is configured to store and retrieve application data residing in the memory **325** and storage **330**. The interconnect **320** is configured to move data, such as programming instructions and application data, between the CPU **305**, I/O devices interface **310**, storage unit **330**, network interface **305**, and memory **325**. Like CPU **205**, CPU **305** is included to be representative of a single CPU, multiple CPUs, a single CPU having multiple processing cores, and the like. Memory **325** is generally included to be representative of a random access memory. The network interface **315** is configured to transmit data via the communications network **120**. Although shown as a single unit, the storage **330** may be a combination of fixed and/or removable storage devices, such as fixed disc drives, removable memory cards, optical storage, SSD or flash memory devices, network attached storage (NAS), or connections to storage area-network (SAN) devices.

[0042] As shown, the memory **325** stores a stream debugger **335**, a stream compiler **300** and a stream manager **132**. And the storage **330** includes the operator graph **134**. As noted above, the stream debugger may be used to manage the debugging of a processing element (PE) on one (or more) of the compute nodes **105**. For example, the stream debugger **335** may determine that one processing node should be fused (or un-fused) with other PEs. In such a case, the stream debugger could determine (from the operator graph) that a large group of PEs have been fused on one compute node—but that debug breakpoints are included in only few of the processing elements. In such a case, the stream debugger **335** could decide to un-fuse the PEs with the breakpoints from the larger group. In one embodiment, the stream debugger **335** un-fuses the PE by invoking the stream compiler **340** to recompile elements of the PE/operator source code as needed. In the current example, the PEs with breakpoints could be recompiled into a first PE, and the remaining PEs (from the fused PE) are recompiled into a second PE. Once prepared, the stream manager **132** may remove the fused PE from the running stream and replace it with the first and second PE generated by the stream compiler **340**. Thereafter, breakpoints in the first PE are hit, the first PE stops executing without disrupting any processing performed by the second PE.

[0043] FIG. **4** illustrates an example of compute nodes in a stream application, according to one embodiment of the invention. As shown, a fused processing element **405** on compute node **130**$_2$ includes three processing elements (labeled PE**1**-PE**3**), which receives an N-tuple data stream and emits an N-tuple to a processing element **410** (labeled PE**4**) on compute node **130**$_3$. In this example, the fused processing element **405** includes source PE **135** (labeled PE**1**), which receives a tuple which includes <name, department, salary, gender>. PE**1** takes this N-tuple and generates one set of tuples sent to PE**2** and another set of tuples sent to PE**3**, based on the gender value in a tuple received by PE**1**. In turn, PE**2** and PE**3** perform a database write for each tuple received from PE**1** and generate a tuple of <name, department, salary> sent to PE**4**. Once received, PE**4** accesses a third-party web service and generates a tuple sent further downstream in the stream application.

[0044] FIG. **5** illustrates a method **500** for debugging a processing element in a stream database application, according to one embodiment of the invention. As shown, the method **500** begins at step **505**, where a debugger application receives an indication of a breakpoint for a processing element included in an operator graph of a stream application. At step **510**, the debugger application may identify a position of the processing element in the operator graph, relative to other processing elements. Based on the position, the debugger application may determine to fuse (or un-fuse) processing elements in the operator graph (step **515**). Additionally, the debugging application may make other changes to the operation of the stream application, e.g., by duplicating a processing element with the breakpoint or exempting certain data tuples (or data flows) from being debugged. Similarly, the debugging application could specify that certain processing elements should be blocked from sending (or receiving) data tuples or should load shed the results of data processing generated during a debugging session.

[0045] At step **520**, any PEs that has been fused or un-fused may be restarted and deployed to the application stream (including the PE with the breakpoint). At step **525**, once the PEs are running in the application stream (and any other changes the debugger selected to make to the stream application), the operations of the PE under debug may be evaluated. For example, the PE under debug may halt execution when a breakpoint is triggered, allowing a developer to then execute the PE in a step-wise manner by stepping into (or over) function calls, to inspect variables and to perform any other debugging functions to evaluate the processing element. At the same time, data tuples continue to flow to the PE being debugged and may be stored in a buffer. That is, tuples flowing to the PE being debugged may be buffered once a breakpoint is triggered until the PE resumes execution.

[0046] At step **530**, once the debugging session is complete, the debugging application may restore the state of the stream application, undoing any changes made to support the debugging session. Accordingly, PEs fused (or un-fused) as part of the debugging session may be un-fused (or re-fused), tuple flows may be restored or buffered data results may be load shed, duplicated processing elements may be removed, etc.

[0047] FIG. **6-8** provide examples of the compute nodes and stream application first shown in FIG. **4** modified to illustrate scenarios for debugging the stream application, according to embodiments of the invention. First, FIG. **6** illustrates an example of the fused processing element **405** of FIG. **4** after being partially un-fused to allow one of the processing elements to be debugged. Assume that a developer adds one or more breakpoints in PE**2**. In such a case, the debugger application could determine that PE**2** can be debugged independently from the processing performed by PE**1** and PE**3** and un-fuse PE**2** from the fused processing element **405**. Accordingly, as shown, fused PE **410** has been modified to remove PE**2**, resulting in partially un-fused PE **405'**. Additionally, PE**2** is now executing as un-fused PE **605**. Running PE**2** as an independent process allows PE**2** to be debugged without slowing down the application stream as data tuples flow through partially un-fused PE **405'**. At the same time, the basic structure of the operator graph remains unchanged; tuples still flow first to PE**1**, which generates tuples sent to PE**2** and PE**3**.

[0048] Similarly, FIG. **7** illustrates an example of the processing element **410** of FIG. **4** after being fused with additional PEs. In this example, assume a user places a breakpoint in PE**4**. In such a case, the debugger application could determine that a downstream processing element (PE**5**) has a dependency such that if PE**4** stops executing PE**5** should stop executing as well. For example, PE**5** may be configured to receive data from PE**4** regularly specified intervals. Given this dependency, when the user inserts a breakpoint in PE**4**, the debugger application may fuse PE**4** and PE**5** into a fused processing element **410'**. Doing so allows PE**4** to be debugged without disrupting the operations of PE**5**, as when the breakpoint is hit in PE**4**, the operations of both PE**4** and PE**5** stop executing.

[0049] FIG. **8** illustrates another example of modifying the processing element **410** of FIG. **4**. In this example, a user again places a breakpoint in PE **410** (labeled PE**4**). However, in this case assume that the debugger application determines that debugging PE **410** should not disrupt data flowing through the stream to PE **805** (labeled PE**5**). In such a case, the debugger application could create debug clone **410'** and insert it into the application stream. This result is shown in FIG. **8**, where the data flowing from PE **405** is duplicated and sent to both PE **410** and debug clone **410'**. Thus, PE **410** continues to process the stream and emit tuples into the stream received by the PE **805**. At the same time, data flowing from PE **405** is also sent to debug clone **410** and outputs tuples to a debug log. Doing so allows the debugger to run a debugging session for PE **410'** without disrupting the application stream.

[0050] Advantageously, embodiments of the invention described above provide techniques for debugging a processing element (or elements) in a stream based application. In particular, embodiments of the invention provide techniques for debugging a processing element in a manner that reduces the impact of debugging the processing element (or elements) on the overall running environment by selectively fusing (or un-fusing) processing elements running on a group of compute nodes. In addition to fusing and un-fusing processing elements, a debugging application and stream manager may modify data flows within the application stream in a variety of ways to minimize any disruption resulting from a debugging session.

[0051] The flowchart and block diagrams in the Figures illustrate the architecture, functionality and operation of possible implementations of systems, methods and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

[0052] While the foregoing is directed to embodiments of the present invention, other and further embodiments of the invention may be devised without departing from the basic scope thereof, and the scope thereof is determined by the claims that follow.

What is claimed is:

1. A method of debugging a stream application composed from a plurality of processing elements executing on one or more compute nodes, the method, comprising:

receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements;

evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality; and

modifying a state of the stream application to allow the first processing element to be debugged; and

initiating a debug session for the first processing element.

2. The method of claim **1**, wherein each processing element executes as a computing process allocated a process ID (PID) on a given one of the compute nodes.

3. The method of claim **1**, wherein the first processing element is configured to receive a first N-tuple, process the first N-tuple, and generate a second N-tuple emitted to a second processing element.

4. The method of claim **1**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:

generating a debug clone of the first processing element;

inserting the breakpoint in the debug clone; and

inserting the debug clone into the stream application.

5. The method of claim **1**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:

halting the execution of at least a second processing element;

generating a fused processing element from the first and second processing elements;

inserting the breakpoint in the fused processing element; and

inserting the fused processing element into the stream application.

6. The method of claim **1**, wherein the first processing element is a fused processing element that includes the first processing element and a second processing element, and wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:

halting the execution of the first processing element;

generating a first un-fused processing element that includes the first processing element;

inserting the breakpoint in the first un-fused processing element;

generating a first un-fused processing element that includes at least the second processing element;

inserting the first and second un-fused processing elements into the stream application.

7. The method of claim **1**, further comprising:

following the debugging session, restoring the state of the stream application.

8. A computer-readable storage medium containing a program which, when executed, performs an operation for debugging a stream application composed from a plurality of processing elements executing on one or more compute nodes, the operation, comprising:

receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements;

evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality; and

modifying a state of the stream application to allow the first processing element to be debugged; and

initiating a debug session for the first processing element.

9. The computer-readable storage medium of claim **8**, wherein each processing element executes as a computing process allocated a process ID (PID) on a given one of the compute nodes.

10. The computer-readable storage medium of claim **8**, wherein the first processing element is configured to receive a first N-tuple, process the first N-tuple, and generate a second N-tuple emitted to a second processing element.

11. The computer-readable storage medium of claim **8**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:

generating a debug clone of the first processing element;
inserting the breakpoint in the debug clone; and
inserting the debug clone into the stream application.

12. The computer-readable storage medium of claim **8**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:
halting the execution of at least a second processing element;
generating a fused processing element from the first and second processing elements;
inserting the breakpoint in the fused processing element; and
inserting the fused processing element into the stream application.

13. The computer-readable storage medium of claim **8**, wherein the first processing element is a fused processing element that includes the first processing element and a second processing element, and wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:
halting the execution of the first processing element;
generating a first un-fused processing element that includes the first processing element;
inserting the breakpoint in the first un-fused processing element;
generating a first un-fused processing element that includes at least the second processing element;
inserting the first and second un-fused processing elements into the stream application.

14. The computer-readable storage medium of claim **8**, wherein the operation further comprises:
following the debugging session, restoring the state of the stream application.

15. A system, comprising:
a plurality of compute nodes, each comprising a processor and a memory, wherein the compute nodes are configured to execute processing elements of a stream application; and
a management system comprising at least a processor and a memory, wherein the memory stores a stream debugging application, which, when executed on the management system, is configured to perform an operation for debugging the stream application executing on the plurality of compute nodes, the operation, comprising:
receiving an indication of a location for a debugging breakpoint in a first one of the plurality of processing elements,
evaluating an operator graph to identify a position of the first processing element, relative to one or more other processing elements, of the plurality,
modifying a state of the stream application to allow the first processing element to be debugged, and
initiating a debug session for the first processing element.

16. The system of claim **15**, wherein each processing element executes as a computing process allocated a process ID (PID) on a given one of the compute nodes.

17. The system of claim **15**, wherein the first processing element is configured to receive a first N-tuple, process the first N-tuple, and generate a second N-tuple emitted to a second processing element.

18. The system of claim **15**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:
generating a debug clone of the first processing element;
inserting the breakpoint in the debug clone; and
inserting the debug clone into the stream application.

19. The system of claim **15**, wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:
halting the execution of at least a second processing element;
generating a fused processing element from the first and second processing elements;
inserting the breakpoint in the fused processing element; and
inserting the fused processing element into the stream application.

20. The system of claim **15**, wherein the first processing element is a fused processing element that includes the first processing element and a second processing element, and wherein modifying the state of the stream application to allow the first processing element to be debugged, comprises:
halting the execution of the first processing element;
generating a first un-fused processing element that includes the first processing element;
inserting the breakpoint in the first un-fused processing element;
generating a first un-fused processing element that includes at least the second processing element;
inserting the first and second un-fused processing elements into the stream application.

21. The system of claim **15**, wherein the operation further comprises:
following the debugging session, restoring the state of the stream application.

* * * * *